

PL/pgSQL Динамические команды



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Причины использования

Выполнение динамического запроса

Способы формирования динамического запроса

Текст SQL-команды формируется в момент выполнения

Причины использования

- дополнительная гибкость в приложении
- формирование нескольких конкретных запросов вместо одного универсального для оптимизации

Цена

- операторы не подготавливаются
- возрастает риск внедрения SQL-кода
- возрастает сложность сопровождения

Под динамическими командами понимаются команды SQL, текст которых формируется и затем выполняется внутри PL/pgSQL-блока в подпрограммах или в анонимных блоках.

В большинстве случаев можно обойтись без динамических команд, но иногда они могут предоставить дополнительную гибкость. Например, можно встроить в определенные места приложения возможность выполнять команды, считанные из настроек системы. Управлять такими настройками могут специалисты поддержки во время эксплуатации приложения, а не программисты в момент разработки.

Иногда, при формировании отчета с большим количеством необязательных параметров, бывает проще формировать текст запроса прямо во время выполнения только для указанных параметров, чем заранее, при разработке, писать сложный запрос, учитывающий все возможные комбинации параметров.

Платой за использование динамических команд будет отказ от подготовленных операторов, которые по умолчанию используются в PL/pgSQL. Также нужно следить за безопасностью динамических команд с точки зрения возможности внедрения SQL-кода.

Следует отметить и существенное возрастание сложности сопровождения. В частности, в исходном коде приложения невозможно будет текстовым поиском обнаружить все выполняемые команды.

Оператор EXECUTE

- выполняет строковое представление SQL-запроса
- позволяет использовать параметры
- переменные PL/pgSQL не становятся неявными параметрами

Может использоваться вместо SQL-запроса

- сам по себе
- при открытии курсора
- в цикле по запросу
- в предложении RETURN QUERY

Для выполнения динамических команд в PL/pgSQL используется команда EXECUTE, которая выполняет SQL-оператор, заданный в виде текстовой строки.

Динамический запрос может содержать явно заданные параметры. В тексте команды параметры обозначаются как \$1, \$2 и т. д., а значения параметров указываются в предложении USING. Параметры работают точно так же, как в подготовленных операторах (что рассматривалось в теме «Архитектура. Общее устройство PostgreSQL»). Однако переменные PL/pgSQL не становятся неявными параметрами, как это происходит при обычном, не динамическом, использовании SQL в PL/pgSQL.

Оператор EXECUTE может использоваться как самостоятельный оператор (в этом случае просто выполняется динамическая команда). Он также может использоваться в циклах по запросу, при открытии курсора, в команде RETURN QUERY — во всех этих случаях EXECUTE заменяет собой оператор SQL.

<https://postgrespro.ru/docs/postgresql/16/plpgsql-statements#PLPGSQL-STATEMENTS-EXECUTING-DYN>

Тонкий момент: процедура не может управлять транзакциями, если она вызывается оператором EXECUTE.

Выполнение динамического запроса

Оператор EXECUTE позволяет выполнить SQL-команду, заданную в виде строки.

```
=> DO $$
DECLARE
    cmd CONSTANT text := 'CREATE TABLE city_msk(
        name text, architect text, founded integer
    )';
BEGIN
    EXECUTE cmd; -- таблица для исторических зданий Москвы
END
$$;

DO
```

Предложение INTO оператора EXECUTE позволяет вернуть одну (первую) строку результата в переменную составного типа или несколько скалярных переменных.

Для проверки результата выполнения динамической команды можно использовать команду GET DIAGNOSTICS, как и в случае статических команд (но не переменную FOUND).

```
=> DO $$
DECLARE
    rec record;
    cnt bigint;
BEGIN
    EXECUTE 'INSERT INTO city_msk (name, architect, founded) VALUES
        (''Пашков дом'', ''Василий Баженов'', 1784),
        (''Ансамбль Царицыно'', ''Василий Баженов'', 1776),
        (''Усадьба Туголмина'', ''Василий Баженов'', 1788),
        (''Музей Пушкина'', ''Роман Клейн'', 1898),
        (''ЦУМ'', ''Роман Клейн'', 1908)
    RETURNING name, architect, founded'
    INTO rec;
    RAISE NOTICE '%', rec;
    GET DIAGNOSTICS cnt := ROW_COUNT;
    RAISE NOTICE 'Добавлено строк: %', cnt;
END
$$;

NOTICE:  ("Пашков дом", "Василий Баженов", 1784)
NOTICE:  Добавлено строк: 5
DO
```

При необходимости с помощью STRICT можно гарантировать, что команда обработает ровно одну строку.

Результат динамического запроса можно обработать в цикле FOR.

```
=> DO $$
DECLARE
    rec record;
BEGIN
    FOR rec IN EXECUTE 'SELECT * FROM city_msk WHERE architect = ''Роман Клейн'' ORDER BY founded'
    LOOP
        RAISE NOTICE '%', rec;
    END LOOP;
END
$$;

NOTICE:  ("Музей Пушкина", "Роман Клейн", 1898)
NOTICE:  (ЦУМ, "Роман Клейн", 1908)
DO
```

Этот же пример с использованием курсора.

```

=> DO $$
DECLARE
    cur refcursor;
    rec record;
BEGIN
    OPEN cur FOR EXECUTE 'SELECT * FROM city_msk WHERE architect = ''Роман Клейн'' ORDER BY founded';
    LOOP
        FETCH cur INTO rec;
        EXIT WHEN NOT FOUND;
        RAISE NOTICE '%', rec;
    END LOOP;
END
$$;

NOTICE:  ("Музей Пушкина", "Роман Клейн", 1898)
NOTICE:  (ЦУМ, "Роман Клейн", 1908)
DO

```

Оператор RETURN QUERY для возврата строк из функции также может использовать динамические запросы. Напишем функцию, возвращающую все здания, возведенные архитектором до определенной даты. Для этого нам понадобятся параметры:

```

=> CREATE FUNCTION sel_msk(architect text, founded integer DEFAULT NULL)
RETURNS SETOF text
AS $$
DECLARE
    -- параметры пронумерованы: $1, $2...
    cmd text := '
        SELECT name FROM city_msk
        WHERE architect = $1 AND ($2 IS NULL OR founded < $2)';
BEGIN
    RETURN QUERY
        EXECUTE cmd
        USING architect, founded; -- указываем значения по порядку
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION

=> SELECT * FROM sel_msk('Роман Клейн');

    sel_msk
-----
Музей Пушкина
ЦУМ
(2 rows)

=> SELECT * FROM sel_msk('Роман Клейн', 1905 ); -- до событий на Красной Пресне

    sel_msk
-----
Музей Пушкина
(1 row)

```

Подстановка значений параметров

предложение USING

гарантируется невозможность внедрения SQL-кода

Экранирование значений

идентификаторы: `format('%I')`, `quote_ident`

литералы: `format('%L')`, `quote_literal`, `quote_nullable`

внедрение SQL-кода невозможно при правильном использовании

Обычные строковые функции

конкатенация и др.

возможно внедрение SQL-кода!

Использование оператора EXECUTE имеет смысл, если команда формируется динамически. Приведенные ранее примеры можно было бы записать и без оператора EXECUTE.

Поскольку команда представляется текстовой строкой, ее можно сформировать обычными строковыми функциями, такими как конкатенация и т. п. В этом случае надо проявлять крайнюю осторожность, поскольку возможно внедрение SQL-кода.

Внедрение SQL-кода невозможно в принципе, если значения передаются как параметры с помощью предложения USING.

Однако параметры применимы не всегда: может понадобиться конкатенировать отдельные части запроса или подставлять в запрос имя таблицы. В этом случае для защиты от внедрений следует экранировать значения, полученные из ненадежного источника.

Для идентификаторов используется функция `format` со спецификатором `%I` или функция `quote_ident`. Эти функции формируют правильные имена идентификаторов, при необходимости заключая их в двойные кавычки и экранируя специальные символы.

Для подстановки литералов внутрь текста команды можно использовать функции `quote_literal`, `quote_nullable` или функцию `format` со спецификатором `%L`.

<https://postgrespro.ru/docs/postgresql/16/functions-string>

Возможность внедрения SQL-кода

Перепишем функцию, возвращающую здания, добавив параметр — код города. По задумке такая функция должна позволять обращаться только к таблицам, начинающимся на city_.

```
=> CREATE FUNCTION sel_city(  
    city_code text,  
    architect text,  
    founded integer DEFAULT NULL  
)  
RETURNS SETOF text AS $$  
DECLARE  
    cmd text := '  
        SELECT name FROM city_' || city_code || '  
        WHERE architect = $1 AND ($2 IS NULL OR founded < $2)';  
BEGIN  
    RAISE NOTICE '%', cmd;  
    RETURN QUERY  
        EXECUTE cmd  
        USING architect, founded;  
END  
$$ LANGUAGE plpgsql;  
  
CREATE FUNCTION
```

Функция правильно работает при «нормальных» значениях параметров:

```
=> SELECT * FROM sel_city('msk', 'Василий Баженов');  
  
NOTICE:  
      SELECT name FROM city_msk  
      WHERE architect = $1 AND ($2 IS NULL OR founded < $2)  
sel_city  
-----  
Пашков дом  
Ансамбль Царицыно  
Усадьба Тутолмина  
(3 rows)
```

Однако злоумышленник может подобрать такое значение, которое изменит синтаксическую конструкцию запроса и позволит ему получить несанкционированный доступ к данным:

```
=> SELECT * FROM sel_city('msk WHERE false  
    UNION ALL  
    SELECT username FROM pg_user  
    UNION ALL  
    SELECT name FROM city_msk', '');  
  
NOTICE:  
      SELECT name FROM city_msk WHERE false  
      UNION ALL  
      SELECT username FROM pg_user  
      UNION ALL  
      SELECT name FROM city_msk  
      WHERE architect = $1 AND ($2 IS NULL OR founded < $2)  
sel_city  
-----  
student  
postgres  
(2 rows)
```

При использовании подготовленных операторов или динамических команд с параметрами это невозможно в принципе, так как структура SQL-запроса фиксируется при синтаксическом разборе. Выражение всегда останется выражением и не сможет превратиться, например, в имя таблицы.

Формирование динамической команды

Параметры в предложении USING нельзя использовать для имен объектов (названия таблиц, столбцов и пр.) в динамической команде. Такие идентификаторы необходимо экранировать, чтобы структура запроса не могла измениться:


```
=> SELECT format('%I', 'foo'),
         format('%I', 'foo bar'),
         format('%I', 'foo"bar');

format | format | format
-----+-----+-----
foo    | "foo bar" | "foo"bar"
(1 row)
```

То же самое выполняет и другая функция:

```
=> SELECT quote_ident('foo'),
         quote_ident('foo bar'),
         quote_ident('foo"bar');

quote_ident | quote_ident | quote_ident
-----+-----+-----
foo         | "foo bar"   | "foo"bar"
(1 row)
```

Вот как может выглядеть пример с созданием таблицы:

```
=> DO $$
DECLARE
    cmd CONSTANT text := 'CREATE TABLE %I(
        name text, architect text, founded integer
    )';
BEGIN
    EXECUTE format(cmd, 'city_spb'); -- таблица для Санкт-Петербурга
    EXECUTE format(cmd, 'city_nov'); -- таблица для Новгорода
END
$$;

DO
```

Вместо использования параметров, можно вставлять в строку литералы. В этом случае также требуется экранирование, но другое:

```
=> SELECT format('%L', 'foo bar'),
         format('%L', 'foo''bar'),
         format('%L', NULL);

format | format | format
-----+-----+-----
'foo bar' | 'foo''bar' | NULL
(1 row)
```

Это же выполняет и функция quote_nullable:

```
=> SELECT quote_nullable('foo bar'),
         quote_nullable('foo''bar'),
         quote_nullable(NULL);

quote_nullable | quote_nullable | quote_nullable
-----+-----+-----
'foo bar'      | 'foo''bar'     | NULL
(1 row)
```

Похожая функция quote_literal отличается тем, что не превращает неопределенное значение в литерал:

```
=> SELECT quote_literal(NULL);

quote_literal
-----
(1 row)
```

В качестве примера перепишем функцию, возвращающую здания в определенном городе, без использования параметров, но так, чтобы она осталась безопасной.

```

=> CREATE OR REPLACE FUNCTION sel_city(
    city_code text,
    architect text,
    founded integer DEFAULT NULL
)
RETURNS SETOF text
AS $$
DECLARE
    cmd text := '
        SELECT name FROM %I
        WHERE architect = %L AND (%L IS NULL OR founded < %L::integer)';
BEGIN
    RETURN QUERY EXECUTE format(
        cmd, 'city_' || city_code, architect, founded, founded
    );
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION

```

Обратите внимание, что в этом случае получается два лишних приведения типов: сначала параметр типа integer приводится к строке, а затем, на этапе выполнения, строка обратно приводится к integer (в случае использования параметров USING такого не происходит):

```

=> SELECT * FROM sel_city('msk', 'Василий Баженов', 1785); -- до приезда императрицы в Царицыно

 sel_city
-----
Пашков дом
Ансамбль Царицыно
(2 rows)

```

Попытка передачи некорректного значения не приведет к успеху:

```

=> SELECT * FROM sel_city('msk WHERE false
    UNION ALL
    SELECT username FROM pg_user
    UNION ALL
    SELECT name FROM city_msk', '');

NOTICE:  identifier "city_msk WHERE false
    UNION ALL
    SELECT username FROM pg_user
    UNION ALL
    SELECT name FROM city_msk" will be truncated to "city_msk WHERE false
    UNION ALL
    SELECT username F"
ERROR:  relation "city_msk WHERE false
    UNION ALL
    SELECT username F" does not exist
LINE 2:         SELECT name FROM "city_msk WHERE false
                        ^

QUERY:
    SELECT name FROM "city_msk WHERE false
    UNION ALL
    SELECT username FROM pg_user
    UNION ALL
    SELECT name FROM city_msk"
    WHERE architect = '' AND (NULL IS NULL OR founded < NULL::integer)
CONTEXT:  PL/pgSQL function sel_city(text,text,integer) line 7 at RETURN QUERY

```

Динамические команды дают дополнительную гибкость

Позволяют формировать отдельные запросы для разных значений параметров с целью оптимизации

Не подходят для коротких, частых запросов

Увеличивается сложность поддержки



1. Измените функцию `get_catalog` так, чтобы запрос к представлению `catalog_v` формировался динамически и содержал условия только на те поля, которые заполнены на форме поиска в «Магазине».

Убедитесь, что реализация не допускает возможности внедрения SQL-кода.

Проверьте работу функции в приложении.

1. Скажем, если на форме поиска не заполнены поля «Название книги» и «Имя автора», но установлен флажок «Есть на складе», должен быть сформирован такой, например, запрос:

```
SELECT ... FROM catalog_v WHERE onhand_qty > 0;
```

Учтите, что поиск при такой реализации вовсе не обязательно будет работать эффективнее, но поддерживать его совершенно точно будет сложнее. Поэтому в реальной работе не стоит прибегать к такому приему, если для того нет веских оснований. Тема оптимизации запросов рассматривается в курсе QPT.

1. Функция get_catalog

```
=> CREATE OR REPLACE FUNCTION get_catalog(  
    author_name text,  
    book_title text,  
    in_stock boolean  
)  
RETURNS TABLE(book_id integer, display_name text, onhand_qty integer)  
AS $$  
DECLARE  
    title_cond text := '';  
    author_cond text := '';  
    qty_cond text := '';  
    cmd text;  
BEGIN  
    IF book_title != '' THEN  
        title_cond := format(  
            ' AND cv.title ILIKE %L', '%' || book_title || '%'  
        );  
    END IF;  
    IF author_name != '' THEN  
        author_cond := format(  
            ' AND cv.authors ILIKE %L', '%' || author_name || '%'  
        );  
    END IF;  
    IF in_stock THEN  
        qty_cond := ' AND cv.onhand_qty > 0';  
    END IF;  
    cmd := 'SELECT cv.book_id,  
                cv.display_name,  
                cv.onhand_qty  
            FROM   catalog_v cv  
            WHERE  true'  
            || title_cond || author_cond || qty_cond || '  
            ORDER BY display_name';  
    RAISE NOTICE '%', cmd;  
    RETURN QUERY EXECUTE cmd;  
END  
$$ STABLE LANGUAGE plpgsql;  
  
CREATE FUNCTION
```

1. Создайте функцию, которая возвращает строки матричного отчета по функциям в базе данных.
 Столбцы должны содержать имена владельцев функций, строки — названия схем, а ячейки — количество функций данного владельца в данной схеме.
 Как можно вызвать такую функцию?

10

1. Примерный вид результата:

<i>schema</i>	<i>total</i>	<i>postgres</i>	<i>student</i>	<i>...</i>
information_schema	12	12	0	
pg_catalog	2811	2811	0	
public	3	0	3	
...				

Количество столбцов в запросе заранее не известно. Поэтому необходимо сконструировать запрос и затем динамически его выполнить. Текст запроса может быть таким:

```
SELECT pronamespace::regnamespace::text AS schema,
       COUNT(*) AS total
       ,SUM(CASE WHEN proowner = 10 THEN 1 ELSE 0 END) postgres
       ,SUM(CASE WHEN proowner = 16384 THEN 1 ELSE 0 END) student
FROM pg_proc
GROUP BY pronamespace::regnamespace
ORDER BY schema
```

Выделенные строки — динамическая часть — необходимо сформировать дополнительным запросом. Начало и конец запроса — статические.

Столбец `proowner` имеет тип `oid`, для получения имени владельца можно воспользоваться конструкцией `proowner::regrole::text`.

Получение матричного отчета

```
=> CREATE DATABASE plpgsql_dynamic;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_dynamic
```

You are now connected to database "plpgsql_dynamic" as user "student".

Вспомогательная функция для формирования текста динамического запроса:

```
=> CREATE FUNCTION form_query() RETURNS text
AS $$
DECLARE
    query_text text;
    columns text := '';
    r record;
BEGIN
    -- Статическая часть запроса.
    -- Первые два столбца: имя схемы и общее количество функций в ней
    query_text :=
$query$
SELECT pronamespace::regnamespace::text AS schema
    , count(*) AS total{{columns}}
FROM pg_proc
GROUP BY pronamespace::regnamespace
ORDER BY schema
$query$;

    -- Динамическая часть запроса.
    -- Получаем список владельцев функций, для каждого — отдельный столбец
    FOR r IN SELECT DISTINCT proowner AS owner FROM pg_proc ORDER BY 1
    LOOP
        columns := columns || format(
            E'\n    , sum(CASE WHEN proowner = %s THEN 1 ELSE 0 END) AS %I',
            r.owner,
            r.owner::regrole
        );
    END LOOP;

    RETURN replace(query_text, '{{columns}}', columns);
END
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Итоговый текст запроса:

```
=> SELECT form_query();
```

```
----- form_query -----
SELECT pronamespace::regnamespace::text AS schema
    , count(*) AS total
    , sum(CASE WHEN proowner = 10 THEN 1 ELSE 0 END) AS postgres
    , sum(CASE WHEN proowner = 16384 THEN 1 ELSE 0 END) AS student
FROM pg_proc
GROUP BY pronamespace::regnamespace
ORDER BY schema

(1 row)
```

Теперь создаем функцию для матричного отчета:

```
=> CREATE FUNCTION matrix() RETURNS SETOF record
AS $$
BEGIN
    RETURN QUERY EXECUTE form_query();
END
$$ STABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

Простое выполнение запроса приведет к ошибке, так как не указана структура возвращаемых записей:

```
=> SELECT * FROM matrix();
```

```
ERROR: a column definition list is required for functions returning "record"
LINE 1: SELECT * FROM matrix();
      ^
```

В этом состоит важное ограничение на использование функций, возвращающих произвольную выборку. В момент вызова необходимо знать и указать структуру возвращаемой записи.

В общем случае структура возвращаемой записи может быть неизвестна, но, применительно к нашему матричному отчету, можно выполнить еще один запрос, который покажет, как правильно вызвать функцию matrix.

Подготовим текст запроса:

```
=> CREATE FUNCTION matrix_call() RETURNS text
AS $$
DECLARE
    cmd text;
    r record;
BEGIN
    cmd := 'SELECT * FROM matrix() AS (
        schema text, total bigint';

    FOR r IN SELECT DISTINCT proowner AS owner FROM pg_proc ORDER BY 1
    LOOP
        cmd := cmd || format(', %I bigint', r.owner::regrole::text);
    END LOOP;
    cmd := cmd || E'\n)';

    RAISE NOTICE '%', cmd;
    RETURN cmd;
END
$$ STABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

Теперь мы можем вызовом matrix_call получить запрос, отражающий структуру матричного отчета, а затем выполнить этот запрос и получить отчет (psql позволяет все это сделать одной командой \gexec):

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> SELECT matrix_call() \gexec
```

```
NOTICE: SELECT * FROM matrix() AS (
    schema text, total bigint, postgres bigint, student bigint
)
 schema      | total | postgres | student
-----+-----+-----+-----
information_schema |    11 |         11 |         0
pg_catalog    |   3286 |        3286 |         0
public        |     3 |          0 |         3
(3 rows)
```

```
=> COMMIT;
```

```
COMMIT
```

Матричный отчет корректно формируется.

- Уровень изоляции Repeatable Read гарантирует, что отчет сформируется, даже если между двумя запросами появится функция у нового владельца.
- Можно было бы и напрямую выполнить запрос, возвращаемый функцией form_query. Но задача получить в клиентском приложении список возвращаемых столбцов все равно останется. Функция matrix_call показывает, как ее можно решить дополнительным запросом.

Еще один вариант решения заключается в том, чтобы вместо набора записей произвольной структуры возвращать набор строк слабоструктурированного типа (такого, как JSON или XML). Эти типы рассматриваются в курсе DEV2.

```
=> \c postgres
```

You are now connected to database "postgres" as user "student".

```
=> DROP DATABASE plpgsql_dynamic;
```

```
DROP DATABASE
```